SMART CONTRACT AUDIT REPORT

for

Carbon Protocol

Prepared By: Xiaomi Huang

PeckShield

May 29, 2024

Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	Carbon Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 29, 2024	Xuxian Jiang	Final Release
1.0-rc	May 18, 2024	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Carbon` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Carbon

The `Carbon` protocol is a next-generation `AMM` which introduces the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of-the-money limit order. `Carbon` offers unique trading abilities which enable adjustable trading strategies with custom ranges, linked orders, and continuous one-directional liquidity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Carbon Protocol

Item	Description
Issuer	Bancor
Website	https://www.carbondefi.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 29, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/bancorprotocol/carbon-contracts.git (d06152e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/bancorprotocol/carbon-contracts.git (64d6dfc)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Carbon` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	2	■ ■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Precision in CarbonPOL::expectedTradeInput()	Coding Practices	Resolved
PVE-002	Informational	Suggested Adherence of Checks-Effects-Interactions	Coding Practices	Resolved
PVE-003	Informational	Revisited _trade() Logic in Strategies	Business Logic	Resolved
PVE-004	Medium	Trust on Admin Keys	Security Features	Mitigated

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Precision in CarbonPOL::expectedTradeInput()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CarbonPOL`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the the preferred handling when calculating the expected trade input amount.

In particular, we show below the implementation of the `CarbonPOL::expectedTradeInput()` routine. It has a rather straightforward logic in calculating the trade input based on the current price. However, the resulting amount is computed via the `MathEx.mulDivF()` helper (line 271), which is in favor of the trading user. We suggest the calculation should be computed in favor of the protocol with `MathEx.mulDivC()`. Though the resulting precision loss may be just a small number, it might play a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

```
262    function expectedTradeInput(Token token, uint128 targetAmount) public view
           validToken(token) returns (uint128) {
263        // revert if not enough amount available for trade
264        if (targetAmount > _amountAvailableForTrading(token)) {
265            revert InsufficientAmountForTrading();
266        }
267        Price memory currentPrice = tokenPrice(token);
268        // revert if current price is not valid
```

```
        _validPrice(currentPrice);
        // calculate the trade input based on the current price
        return MathEx.mulDivF(currentPrice.sourceAmount, targetAmount, currentPrice.
            targetAmount).toUint128();
    }
```

Listing 3.1: `CarbonPOL::expectedTradeInput()`

Recommendation Revise the above calculations in favor of the protocol when facing a possible precision loss.

Status The issue has been resolved as the team plans to completely replace the `CarbonPOL` contract with `CarbonVortex`, which solves the issue in the following PR: 146.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CarbonPOL`, `CarbonVortex`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `CarbonPOL` as an example, the `_sellETHForBNT()` function (see the code snippet below) is provided to externally call `msg.sender` to transfer the `Ether`. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 345) starts before effecting the update on internal states (lines 348 and 356 – 357), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be abused of launching `re-entrancy` via the same entry function. Fortunately, the use of `nonReentrant` effectively blocks this specific risk.

```
    function _sellETHForBNT(uint128 targetAmount) private returns (uint128) {
        uint128 sourceAmount = expectedTradeInput(NATIVE_TOKEN, targetAmount);
        // revert if trade requires 0 bnt
        if (sourceAmount == 0) {
            revert InvalidTrade();
        }
        // transfer the tokens from the user to the bnt address (burn them directly)
        _bnt.safeTransferFrom(msg.sender, Token.unwrap(_bnt), sourceAmount);

        // transfer the eth to the user
        payable(msg.sender).sendValue(targetAmount);

        // update the available eth sale amount
        _ethSaleAmount.current -= targetAmount;

        // check if remaining eth sale amount is below the min eth sale amount
        if (_ethSaleAmount.current < _minEthSaleAmount) {
            // top up the eth sale amount
            _ethSaleAmount.current = Math.min(address(this).balance, _ethSaleAmount.
                initial).toUint128();
            // reset the price to double the current one
            Price memory price = tokenPrice(NATIVE_TOKEN);
            _initialPrice[NATIVE_TOKEN] = price;
            _tradingStartTimes[NATIVE_TOKEN] = uint32(block.timestamp);
            // emit price updated event
            emit PriceUpdated({ token: NATIVE_TOKEN, price: price });
        }

        return sourceAmount;
    }
```

Listing 3.2: `CarbonPOL::_sellETHForBNT()`

Recommendation Revisit the above routine to follow the best practice of the `checks-effects-interactions` pattern. In the meantime, we acknowledge that the use of `nonReentrant` effectively blocks this specific risk.

Status The issue has been resolved with the use of `CarbonVortex` to replace the above `CarbonPOL`.

3.3 Revisited _trade() Logic in Strategies

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Strategies`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Carbon` protocol provides an off-chain matching algorithm whereby a trade of a given number of tokens is spread over one or more orders to achieve the most favorable rate of exchange for the trader. The trade is fulfilled by performing a single trade action in each of the matched orders. While examining the actual trade logic, we notice a possible improvement to validate the token for swap is properly transferred in.

To elaborate, we show below the code snippet of the `_trade()` routine, which is used to fulfill the trade. It loops all the given trade actions, finds the target order in each trade action and calculates the source and target amounts for the single trade action. The source and target amounts of each single trade action is calculated and then added together to know exact token amount for transfer in/out. We notice the final token amount to transfer in is achieved by calling the `_validateDepositAndRefundExcessNativeToken()` routine (line 501). When calling this routine, we suggest to set `validateDepositAmount` as `true` to ensure the funds are properly transferred in.

```
469     function _trade(TradeAction[] calldata tradeActions, TradeParams memory params)
            internal {
470         bool isTargetToken0 = params.tokens.target == params.pair.tokens[0];
471 
472         // process trade actions
473         for (uint256 i = 0; i < tradeActions.length; i = uncheckedInc(i)) {...}
474 
475         // apply trading fee
476         uint128 tradingFeeAmount;
477         if (params.byTargetAmount) {
478             uint128 amountIncludingFee = _addFee(params.sourceAmount, params.pair.id);
479             tradingFeeAmount = amountIncludingFee - params.sourceAmount;
480             params.sourceAmount = amountIncludingFee;
481             if (params.sourceAmount > params.constraint) {
482                 revert GreaterThanMaxInput();
483             }
484             _accumulatedFees[params.tokens.source] += tradingFeeAmount;
485         } else {
486             uint128 amountExcludingFee = _subtractFee(params.targetAmount, params.pair.
                    id);
487             tradingFeeAmount = params.targetAmount - amountExcludingFee;
488             params.targetAmount = amountExcludingFee;
```

```
            if (params.targetAmount < params.constraint) {
                revert LowerThanMinReturn();
            }
            _accumulatedFees[params.tokens.target] += tradingFeeAmount;
        }

        // transfer funds
        _validateDepositAndRefundExcessNativeToken(
            params.tokens.source,
            params.trader,
            params.sourceAmount,
            params.txValue,
            false
        );
        _withdrawFunds(params.tokens.target, payable(params.trader), params.targetAmount
            );

        // tokens traded successfully, emit event
        emit TokensTraded({
            trader: params.trader,
            sourceToken: params.tokens.source,
            targetToken: params.tokens.target,
            sourceAmount: params.sourceAmount,
            targetAmount: params.targetAmount,
            tradingFeeAmount: tradingFeeAmount,
            byTargetAmount: params.byTargetAmount
        });
    }
```

Listing 3.3: `Strategies::_trade()`

Moreover, if we examine all possible calls to `validateDepositAndRefundExcessNativeToken()`, all invocations are made with the last parameter as `true`. With that, we can further simplify the logic to avoid this parameter.

Recommendation Add a proper validation for the given trade actions to ensure the funds are properly transferred in.

Status This issue has been resolved as it is part of the design.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Carbon` protocol, there is a special `admin` account (with the `ROLE_ADMIN`). This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, configure parameters, withdraw funds, and upgrade the proxy). Our analysis shows that the `admin` account and other privileged roles need to be scrutinized. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

```
    function setTradingFeePPM(uint32 newTradingFeePPM) external onlyAdmin validFee(
        newTradingFeePPM) {
        _setTradingFeePPM(newTradingFeePPM);
    }
    ...
    function setPairTradingFeePPM(
        Token token0,
        Token token1,
        uint32 newPairTradingFeePPM
    ) external onlyAdmin validFee(newPairTradingFeePPM) {
        Pair memory _pair = _pair(token0, token1);
        _setPairTradingFeePPM(_pair, newPairTradingFeePPM);
    }
```

Listing 3.4: Example Privileged Operations in `CarbonController`

```
    function setRewardsPPM(uint32 newRewardsPPM) external onlyAdmin validFee(
        newRewardsPPM) {
        _setRewardsPPM(newRewardsPPM);
    }
    ...
    function setPriceResetMultiplier(
        uint32 newPriceResetMultiplier
    ) external onlyAdmin greaterThanZero(newPriceResetMultiplier) {
        _setPriceResetMultiplier(newPriceResetMultiplier);
    }
    ...
    function setMinTokenSaleAmountMultiplier(
        uint32 newMinTokenSaleAmountMultiplier
    ) external onlyAdmin greaterThanZero(newMinTokenSaleAmountMultiplier) {
        _setMinTokenSaleAmountMultiplier(newMinTokenSaleAmountMultiplier);
    }
```

```
...
function setPriceDecayHalfLife(
    uint32 newPriceDecayHalfLife
) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {
    _setPriceDecayHalfLife(newPriceDecayHalfLife);
}
...
function setTargetTokenPriceDecayHalfLife(
    uint32 newPriceDecayHalfLife
) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {
    _setTargetTokenPriceDecayHalfLife(newPriceDecayHalfLife);
}
...
function setTargetTokenPriceDecayHalfLifeOnReset(
    uint32 newPriceDecayHalfLife
) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {
    _setTargetTokenPriceDecayHalfLifeOnReset(newPriceDecayHalfLife);
}
...
function setMaxTargetTokenSaleAmount(
    uint128 newMaxTargetTokenSaleAmount
) external onlyAdmin greaterThanZero(newMaxTargetTokenSaleAmount) {
    _setMaxTargetTokenSaleAmount(newMaxTargetTokenSaleAmount);
}
...
function setMinTargetTokenSaleAmount(
    uint128 newMinTargetTokenSaleAmount
) external onlyAdmin greaterThanZero(newMinTargetTokenSaleAmount) {
    _setMinTokenSaleAmount(_targetToken, newMinTargetTokenSaleAmount);
}
...
function disablePair(Token token, bool disabled) external onlyAdmin {
    _setPairDisabled(token, disabled);
}
...
function withdrawFunds(
    Token[] calldata tokens,
    address payable target,
    uint256[] calldata amounts
) external validAddress(target) validateTokens(tokens) nonReentrant onlyAdmin {
    uint256 len = tokens.length;
    if (len != amounts.length) {
        revert InvalidAmountLength();
    }
    for (uint256 i = 0; i < len; i = uncheckedInc(i)) {
        // safe due to nonReentrant modifier (forwards all available gas in case of
            ETH)
        tokens[i].unsafeTransfer(target, amounts[i]);
    }

    emit FundsWithdrawn({ tokens: tokens, caller: msg.sender, target: target,
        amounts: amounts });
```

```
    }
```

Listing 3.5: Example Privileged Operations in `CarbonVortex`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged accounts to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirms they will use a multi-sig account as the admin.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Carbon` protocol, which is a next-generation `AMM` protocol introducing the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of the-money limit order. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.